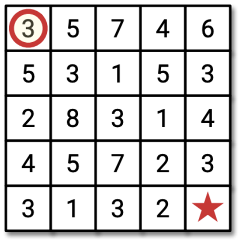
# CSE 40113: Algorithms
## Homework 5

You may complete this homework in groups of 3 or less students. Note that the integrity policy applies: your group should write up your own work, although you're welcome to work on the problems in a larger group. If you have any questions, please re-read both the homework guidelines and the academic integrity policy carefully, and then come discuss any questions or concerns with me.

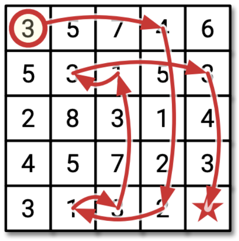
## Required Problems

1. A number maze is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the number maze below, your algorithm should return the integer 8.



| | | | | |
|---|---|---|---|---|
| 3 | 5 | 7 | 4 | 6 |
| 5 | 3 | 1 | 5 | 3 |
| 2 | 8 | 3 | 1 | 4 |
| 4 | 5 | 7 | 2 | 3 |
| 3 | 1 | 3 | 2 | ★ |



   **Solution:** There are two ways to solve this, both fundamentally relying on reductions. For one, you can explicitly build a graph, and then run BFS on it. Alternatively, you can perform a BFS-type search directly on the grid. Both are correct - here, we choose to adapt BFS with "parent" pointers (see page 200 of the book) to run directly on the grid as follows:

```
MinMoves(maze[$n$][$n$]):
  dist[][] ← 2D array of size n x n, initialized to inf everywhere
  dist[0][0] ← 0
  # Use a queue for BFS
  $Q$ ← new empty queue
  Enqueue($Q$, $(0,0)$) # enqueue source node
  while Q is not empty
      $(r, c)$ ← Dequeue($Q$)
      if $(r, c) = (n-1, n-1)$
          return dist[$r$][$c$] # we have reached the goal
      step ← maze[$r$][$c$]
      # List the possible neighbors
      neighbors = [ $(r - step, c), (r + step, c), (r, c - step), (r, c + step)$ ]
      for each $(nr, nc)$ in neighbors
          # Check board bounds
          if $0 \le nr < n$ and $0 \le nc < n$
              # If unvisited, update distance and enqueue
              if dist[$nr$][$nc$] = inf
                  dist[$nr$][$nc$] ← dist[$r$][$c$] + 1
                  Enqueue($Q$, $(nr, nc)$)
      # If we exhaust the queue and never reach $(n-1, n-1)$
      return "No solution"
```

Complexity: We have $n^2$ cells (in this case, nodes). From each cell, we generate one vertex with up to 4 possible neighbors. A BFS on a graph with $V$ nodes and $E$ edges takes $O(V+E)$ time. As we just noted, here $V = n^2$ and $E \le 4n^2$. Thus, the worst-case time complexity is

$$O(n^2 + 4n^2) = O(n^2)$$

Correctness: (via reduction)

In our algorithm, we are treating each cell $(i, j)$ as a vertex, and doing BFS on the resulting graph. As noted in Chapter 5 (see p 202), BFS finds shortest paths in the graphs, and we are adapting the version from page 200 but storing the distance as opposed to the parent. Since the level of a node is always equal to the parent's level +1, this is correctly calculating depth in the DFS tree. Each path in the graph corresponds to a valid sequence of moves in the grid, and likewise each valid sequence of moves will be a path in the graph. Therefore, the shortest sequence of moves will be a shortest path in the graph, which the BFS traversal will find.

■

2. There's a natural (and largely correct!) intuition that two vertices which are far apart in a graph somehow have a more tenuous connection compared to two nodes that are close together. In fact, there are a bunch of algorithmic results that try to make this intuition precise in a variety of ways. In this problem, I want you to consider one of them: namely, how many vertex removals could disconnect the two vertices, so that they cannot reach each other. (This has some obvious implications for reliability of network communication robustness, if you stop to think about it.)

Suppose that we have a graph $G = (V, E)$, and fix two nodes $u$ and $v$. Furthermore, suppose that the distance between $u$ and $v$ is strictly greater than $V/2$. Prove that there must exist some other node $x$ (which is not equal to $u$ or to $v$) whose deletion destroys all $u$ to $v$ paths in the graph. Then, give an algorithm to find such a node (as quickly as possible).

**Solution:** First a proof of why such a vertex must exist: Consider the BFS tree rooted at $u$. We know this tree has depth greater than $V/2$, since the shortest path to $v$ has length greater than $V/2$. There are a total of $V$ vertices, and $u$ is alone at depth 0. We have $> V/2$ levels with $V - 1$ nodes present in these levels, so by the pigeonhole principle we cannot place two per level, as we would run out of vertices in trying to do so. Therefore, some vertex $w$ must be alone and isolated on its level.

Since BFS trees encode shortest paths (see p. 202 again), we know that non-tree edges from the graph can only go between two vertices in adjacent levels or in the same level. (Otherwise, if some edge went from vertex $x$ in level $i$ to vertex $y$ in level $i+2$, we could find a shorter path to $y$ of length $i+1$, which is impossible.). This means that every $u$ to $v$ path must use vertex $w$, since it is the only one in its level. Therefore, we can delete the node $w$, and that will disconnect the graph and destroy every $u$ to $v$ path.

To give an algorithm, we modify again the WFS variant shown on path 200, which stores parent nodes, so that we can find the singleton node on its level as we go. To do this, we track the current level as well as number of vertices on the current level. When the current level goes up, we know the parent node is in the earlier level, so we return the parent if the number of vertices was equal to 1. More formally, fixing the graph $G = (V, E)$, we call the following modification of the WFS from page 200 of the book:

```
BFS(s):
  Initialize all vertices as not marked
  Initialize an empty queue, $Q$
  #process source vertex $s$
  level($s$) $\leftarrow$ 0
  parent($s$) $\leftarrow \emptyset$
  for each edge $(s, v)$
      add $(s, v)$ to $Q$
  #set things up to track how many on a level
  numlevel $\leftarrow$ 0
  currlevel $\leftarrow$ 1
  #Run BFS
  while $Q$ is not empty
      remove $(p, v)$ from $Q$
      if $v$ is unmarked
          mark $v$
          parent($v$) $\leftarrow p$
          level($v$) $\leftarrow$ level($p$) + 1
          # may be at start of a new level: check if found singleton then update
          if currlevel $\neq$ level($v$)
              if numlevel = 1
                  return $p$
              else
                  numlevel $\leftarrow$ 0
          numlevel $\leftarrow$ numlevel +1
          # add neighbors to the queue
          for each edge $\{v, w\}$
              put $(v, w)$ into $Q$
```

Runtime: This is a modified BFS, with only $O(1)$ extra work per dequeue to update levels and check if the parent is the singleton on a level. Hence, the runtime is still $O(V + E)$.

Correctness: Our previous proof showed that such a vertex must exist. So all the remains to do is justify why we are finding it. We are correctly finding parent pointers for the BFS tree, by correctness of the algorithm on page 200 of the book, so here we are correctly storing the level in the tree by setting level of $v$ to be level of $v$'s parent +1. Our if statement then checks if we increment the level number, and if so checks if the previous level only had one node (using the numlevel variable).

Note that is is NOT the only way to solve this, but it is the fastest. For example, you could track a level for each vertex, then find the singleton entry in the vertex level list. However, this naively would require either $O(V \log V + (V + E))$ time to sort and find the singleton. Another option is to try deleting every vertex, then run WFS to see if the graph is disconnected. This would take $O(V(V + E))$ time. Both options are slower, and hence worth partial credit.

■

3. Suppose we are given a directed acyclic graph $G$ whose nodes represent jobs and whose edges represent precedence constraints; that is, each edge $uv$ indicates the job $u$ must be completed

before job v begins. Each node $v$ also has a weight $T(v)$ indicating the time required to execute job $v$. Note that in this problem, jobs can run in parallel.

(a) Describe an algorithm to determine the shortest interval of time in which all jobs in $G$ can be executed.

(b) Suppose the first job starts at time 0. Describe an algorithm to determine, for each vertex v, the earliest time when job $v$ can begin.

(c) Now describe an algorithm to determine, for each vertex v, the latest time when job v can begin without violating the precedence constraints or increasing the overall completion time (computed in part (a)), assuming that every job except $v$ starts at its earliest start time (computed in part (b)).

**Solution:** All three of these can be solved by variations of topological sort, where we propagate finish times from earlier vertices to later ones. For example, assume our input is a graph $G$, a DAG with vertex set $V$ and edge set $E$, along with array $T$, where $T(v)$ is the execution start time of job $v$.

- Parts a and B: We can compute both the shortest interval of time (part a) as well as earliest start time (part b) as follows:

```
# Compute a topological ordering of G: $v_1, \dots v_V$
topoOrder ← TopologicalSort($G$)
# Initialize 2 arrays to hold start and finish times
Initialize startime[1..$V$] and finishTime[1..$V$]
# Compute earliest start times
for $v_i$ in topoOrder
  # All predecessors u of $v_i$ must finish before $v_i$ can start
  maxFinishOfParents ← 0
  for each edge $(u \to v_i)$ in $E$
      if finishTime[$u$] > maxFinishOfParents
          maxFinishOfParents ← finishTime[u]
  # Once all parents are done, v can start
  startime[$v_i$] ← maxFinishOfParents
  finishTime[$v_i$] ← maxFinishOfParents $+T(v_i)$
overallFinish ← $\max_i$\{finishTime[$v_i$] for $v_i$ in $V$\}
```

  Proof of Correctness: The array starttime[$v$] holds the answers to part b, and the array finishTime[$v$] holds the soonest job $v$ can be completed. To prove this, we proceed via induction on the number of times our for loops executes in the topological order.

  Base case: Consider the first iteration. Since we're going in topological order, that first vertex $v_1$ must be a source, with jobs that need to complete earlier. Our for loop will correctly calculate that it can start at time 0 (since there are no incoming edges) and finish at time $T[v_1]$.

  Inductive hypothesis: Assume that the first $k-1$ nodes, $v_1 \dots v_{k-1}$ have correct values for startime and finishTime.

  Inductive Step: Consider vertex $v_k$: since we are running in topological order, we know any edges $v_i \leftarrow v_k$ must come earlier in the ordering, so $i < k$. This means by the inductive hypothesis, all values for start and finish time of the node $v_i$ must be correctly calculated and stored in our arrays. Our code loops over those predecessors, finds the largest finish time among them, and starts $v_k$ running at that time, and stores finish time as that plus $T[v_i]$. If job $k$ started earlier, it would violate one of these predecessors ending before we begin, so we know this is the earliest start time as well as the earliest finish time for $v_k$ (job $k$).

Run time: Topological sort takes $O(V+E)$. The rest of our code loops over every vertex $v_i$, and spends $d(v_i)$ time checking the predecessors, for a total of $\sum_i (1+d(v_i)) = O(V+E)$ using the degree sum formula, so the total time overall is $O(V+E)$.

- For part C: we need to process in reverse topological order. I will assume we have already run the algorithm for parts a and b, so that I have the arrays startTime and finishTime already as well as the topological order $v_1 \dots v_n$.

```
#create an array to store latest start time for each vertex
Initialize lateStart[1..$V$]
for each node $v_k$ in reverse topological order
    if $v_k$ has no outgoing edges
        lateStart[$v_k$] $\leftarrow$ max{startTime[$v_k$], overallFinish$-T[v_k]$}
    else
        lateStart[$v_k$] $\leftarrow$ startTime[$v_k$]
        for each edge $v_k \to u$
            if startTime[$u$] $- T[v_k] >$ lateStart[$v_k$]
                lateStart[$v_k$] $\leftarrow$ startTime[$u$] $- T[v_k]$
```

Runtime: We assume the first algorithm has already been run, taking $O(V+E)$ time, and in this code we loop over every vertex and consider all outgoing edges, doing $O(1)$ work per edge to update times, which again takes $O(V+E)$ time total, so the overall time is still $O(V+E)$.

Correctness: We prove this directly. Consider a vertex $v_k$ in the topological order. If $v_k$ is a sink, it either must start at its earliest start time, if this is the vertex that realizes overall finish time, or else it can start at the overall finish time minus its job length. Any other choice will make the finish time later, so this value is filled in correctly. If $v_k$ is not a sink, then the algorithm considers all outgoing edges $v_k \to v_i$ where $i > k$, and determines what those vertices' earliest start times are, which are correct by our previous proof. It will choose the maximum among all the earliest start times of the $v_i$'s minus $T[v_k]$ to be $v_k$'s latest start. This value is correct because if the job $v_k$ starts any later than this maximum, then the next job $v_i$ would have to begin at a higher time value.

■

4. Sample Solved problem:

   Some friends of yours are working on techniques for coordinating groups of mobile robots. Each robot has a radio transmitter that it uses to communicate with a base station, and your friends find that if the robots get too close to one another, then there are problems with interference among the transmitters. So a natural problem arises: how to plan the motion of the robots in such a way that each robot gets to its intended destination, but in the process the robots don't come close enough together to cause interference problems.

   We can model this problem abstractly as follows. Suppose that we have an undirected graph $G = (V, E)$, representing the floor plan of a building, and there are two robots initially located at nodes $a$ and $b$ in the graph. The robot at node $a$ wants to travel to node $c$ along a path in $G$, and the robot at node $b$ wants to travel to node $d$. This is accomplished by means of a schedule: at each time step, the schedule specifies that one of the robots moves across a single edge, from one node to a neighboring node; at the end of the schedule, the robot from node $a$ should be sitting on $c$, and the robot from $b$ should be sitting on $d$.

   A schedule is interference-free if there is no point at which the two.robots occupy nodes that are at a distance $< r$ from one another in the graph, for a given parameter $r$. We'll assume that the two starting nodes $a$ and $b$ are at a distance greater than $r$, and so are the two ending nodes $c$ and $d$.

   Give a polynomial-time algorithm that decides whether there exists an interference-free schedule by which each robot can get to its destination.

   **Solution:** This is a problem of the following general flavor. We have a set of possible configurations for the robots, where we define a configuration to be a choice of location for each one. We are trying to get from a given starting configuration $(a, b)$ to a given ending configuration $(c, d)$, subject to constraints on how we can move between configurations (we can only change one robot's location to a neighboring node), and also subject to constraints on which configurations are "legal."

   This problem can be tricky to think about if we view things at the level of the underlying graph $G$: for a given configuration of the robots–that is, the current location of each one–it's not clear what rule we should be using to decide how to move one of the robots next. So instead we apply an idea that can be very useful for situations in which we're trying to perform this type of search. We observe that our problem looks a lot like a path-finding problem, not in the original graph $G$ but in the space of all possible configurations.

   So, we will define the following (larger) graph $H$: The node set of $H$ is the set of all possible configurations of the robots; that is, $H$ consists of all possible pairs of nodes in $G$. We join two nodes of $H$ by an edge if they represent configurations that could be consecutive in a schedule; that is, $(u, v)$ and $(u', v')$ will be joined by an edge in $H$ if one of the pairs $u, u'$ or $v, v'$ are equal, and the other pair corresponds to an edge in $G$.

   We can already observe that paths in $H$ from $(a, b)$ to $(c, d)$ correspond to schedules for the robots: such a path consists precisely of a sequence of configurations in which, at each step, one robot crosses a single edge in $G$. However, we have not yet encoded the notion that the schedule should be interference-free.

   To do this, we simply delete from $H$ all nodes that correspond to configurations in which there would be interference. Thus we define $H'$ to be the graph obtained from $H$ by deleting all nodes $(u, v)$ for which the distance between $u$ and $v$ in $G$ is at most $r$.

The full algorithm is then as follows. We construct the graph $H'$, and then run the DFS/BFS algorithm from the text to determine whether there is a path from $(a, b)$ to $(c, d)$. The correctness of the algorithm follows from the fact that paths in $H'$ correspond to schedules, and the nodes in $H'$ correspond precisely to the configurations in which there is no interference.

Finally, we need to consider the running time. Let $V$ denote the number of nodes in $G$, and $E$ denote the number of edges in $G$. We'll analyze the running time by doing three things: (1) bounding the size of $H'$ (which will in general be larger than G), (2) bounding the time it takes to construct H', and (3) bounding the time it takes to search for a path from $(a, b)$ to $(c, d)$ in $H'$.

First, then, let's consider the size of $H'$. $H'$ has at most $V^2$ nodes, since its nodes correspond to pairs of nodes in $G$. Now, how many edges does $H'$ have? A node $(u, v)$ will have edges to $(u', v)$ for each neighbor $u'$ of $u$ in $G$, and to $(u, v')$ for each neighbor $v'$ of $v$ in $G$. A simple upper bound says that there can be at most $V$ choices for $(u', u)$, and at most $V$ choices for $(u, v')$, so there are at most $2V$ edges incident to each node of H'. Summing over the (at most) $V^2$ nodes of $H'$, this would give $O(V^3)$ edges. However, we can also note that $u$ has exactly $d(u)$ neighbors, and $v$ has exactly $d(v)$. Using the degree sum formula, we can slightly improve our total number of edges to $O(VE)$ total.

Second, we bound the time needed to construct $H'$. We first build $H$ by enumerating all pairs of nodes in $G$ in time $O(V^2)$, and constructing edges using the definition above in time $O(V)$ per node, for a total of $O(V^3)$. Next, we must delete nodes from $H$ so as to produce $H'$. We can do this as follows: For each node $u \in G$, we run a breadth-first search from $u$ and identify all nodes $v$ within distance $r$ of $u$. We list all these pairs $(u, v)$ and delete them from $H$. Each breadth-first search in $G$ takes time $O(V + E)$, and we're doing one from each node, so the total time for this part is $O(V(V + E))$.

Finally, we have $H'$, and so we just need to decide whether there is a path from $(a, b)$ to $(c, d)$. This can be done using the connectivity algorithm from the text in time that is linear in the number of nodes and edges of $H'$. Since $H'$ has $O(V^2)$ nodes and $O(VE)$ edges, this final step takes $O(V^2 + VE)$.